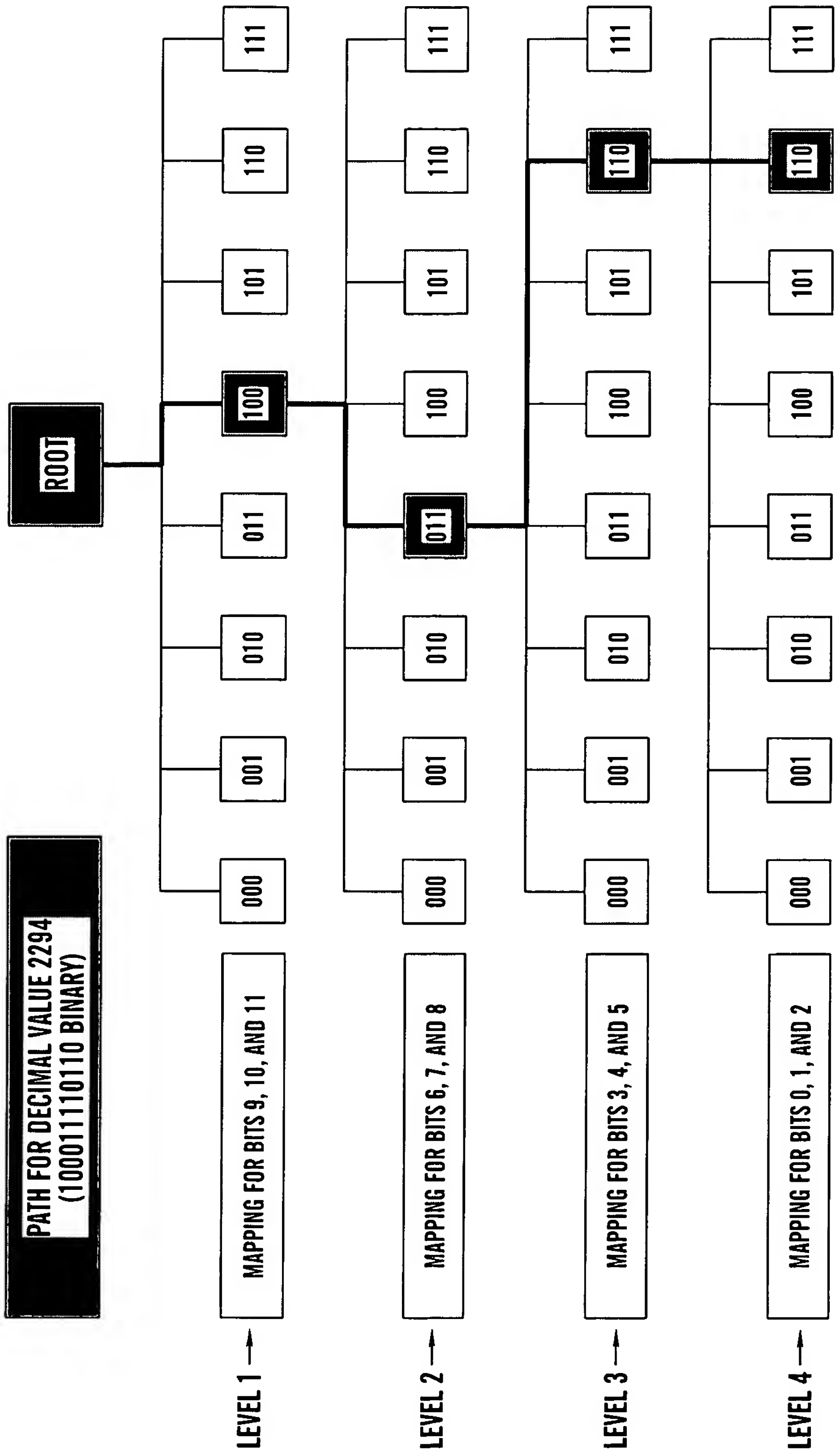

PATH FOR DECIMAL VALUE 2294
(100011110110 BINARY)
ROOT
LEVEL 1
MAPPING FOR BITS 9, 10, AND 11
LEVEL 2
MAPPING FOR BITS 6, 7, AND 8
LEVEL 3
MAPPING FOR BITS 3, 4, AND 5
LEVEL 4
MAPPING FOR BITS 0, 1, AND 2
000
001
010
011
100
101
110
111

FIG. 1

SAMPLE VALUES

DECIMAL	BINARY
12	(001100)
47	(101111)
44	(101100)
37	(100101)
03	(000011)
14	(001110)
31	(011111)
44	(101100)

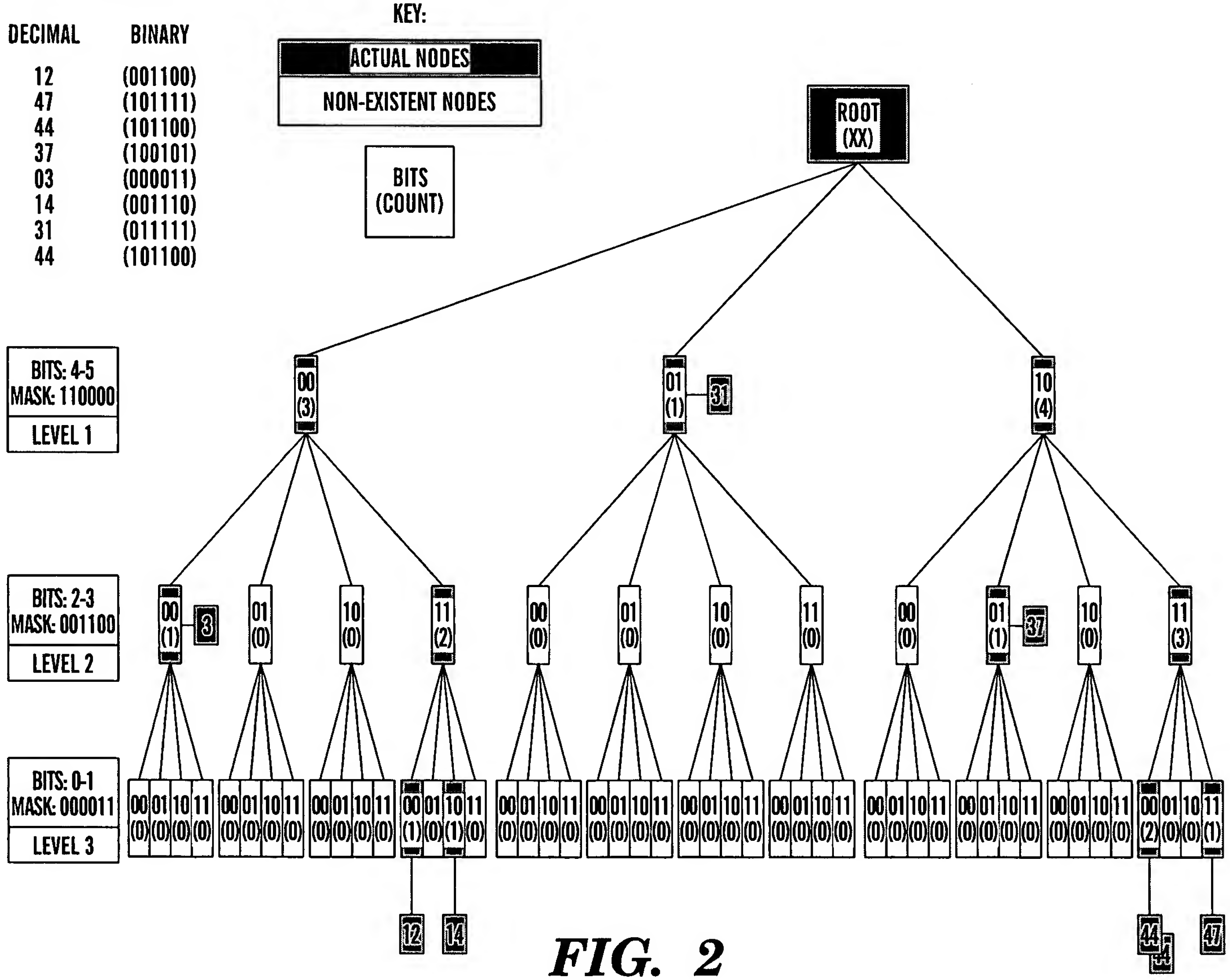


FIG. 2

SAMPLE VALUES

DECIMAL	BINARY
12	(001100)
47	(101111)
44	(101100)
37	(100101)
03	(000011)
14	(110101)
31	(011111)
44	(101100)

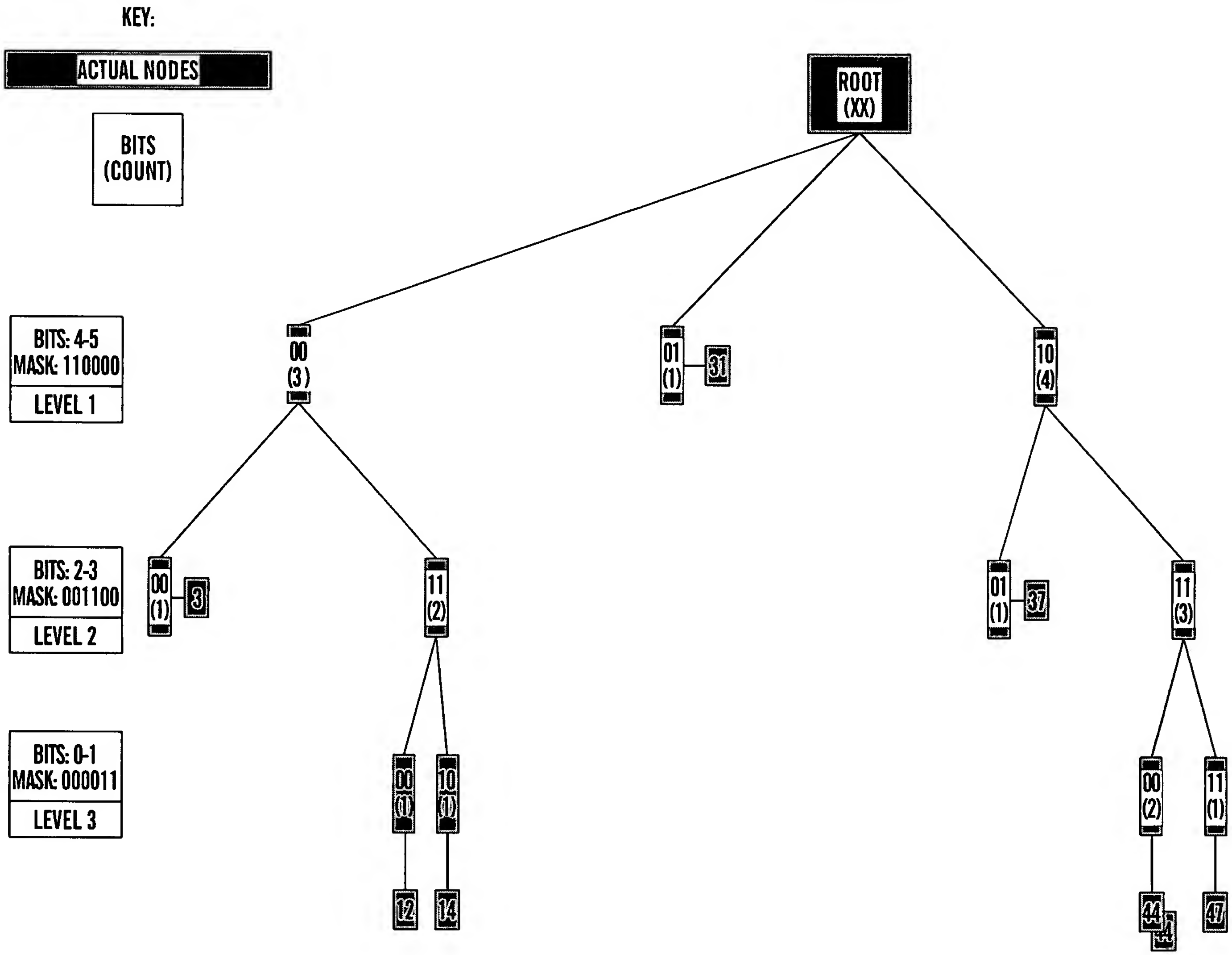


FIG. 3

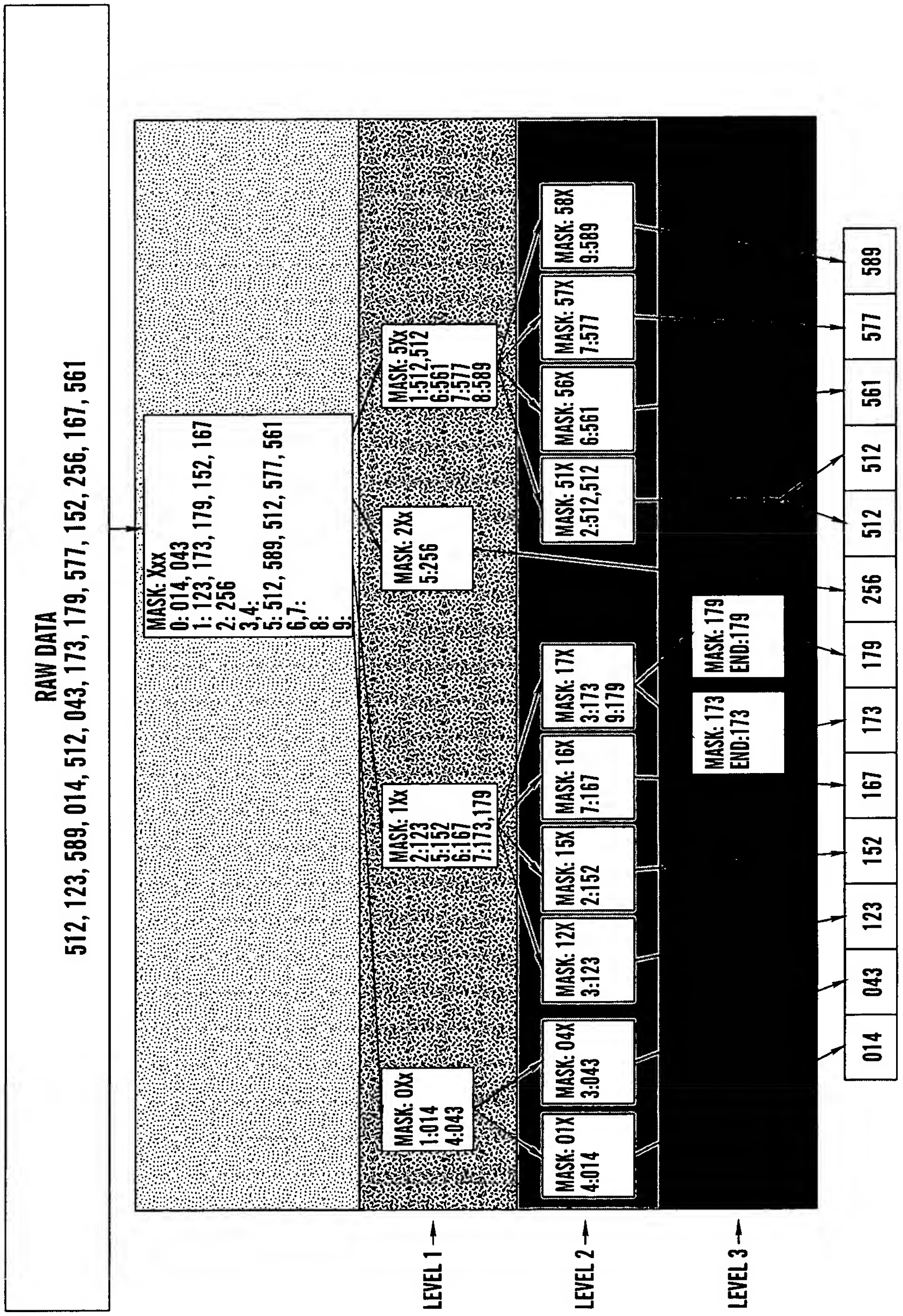

RAW DATA
512, 123, 589, 014, 512, 043, 173, 179, 577, 152, 256, 167, 561
MASK: Xxx
0: 014, 043
1: 123, 173, 179, 152, 167
2: 256
3,4:
5: 512, 589, 512, 577, 561
6,7:
8:
9:
MASK: 0Xx
1:014
4:043
MASK: 1Xx
2:123
5:152
6:167
7:173,179
MASK: 2Xx
5:256
MASK: 5Xx
1:512,512
6:561
7:577
8:589
MASK: 01X
4:014
MASK: 04X
3:043
MASK: 12X
3:123
MASK: 15X
2:152
MASK: 16X
7:167
MASK: 17X
3:173
9:179
MASK: 51X
2:512,512
MASK: 56X
6:561
MASK: 57X
7:577
MASK: 58X
9:589
MASK: 173
END:173
MASK: 179
END:179
014
043
123
152
167
173
179
256
512
512
561
577
589
LEVEL 1
LEVEL 2
LEVEL 3

FIG. 4

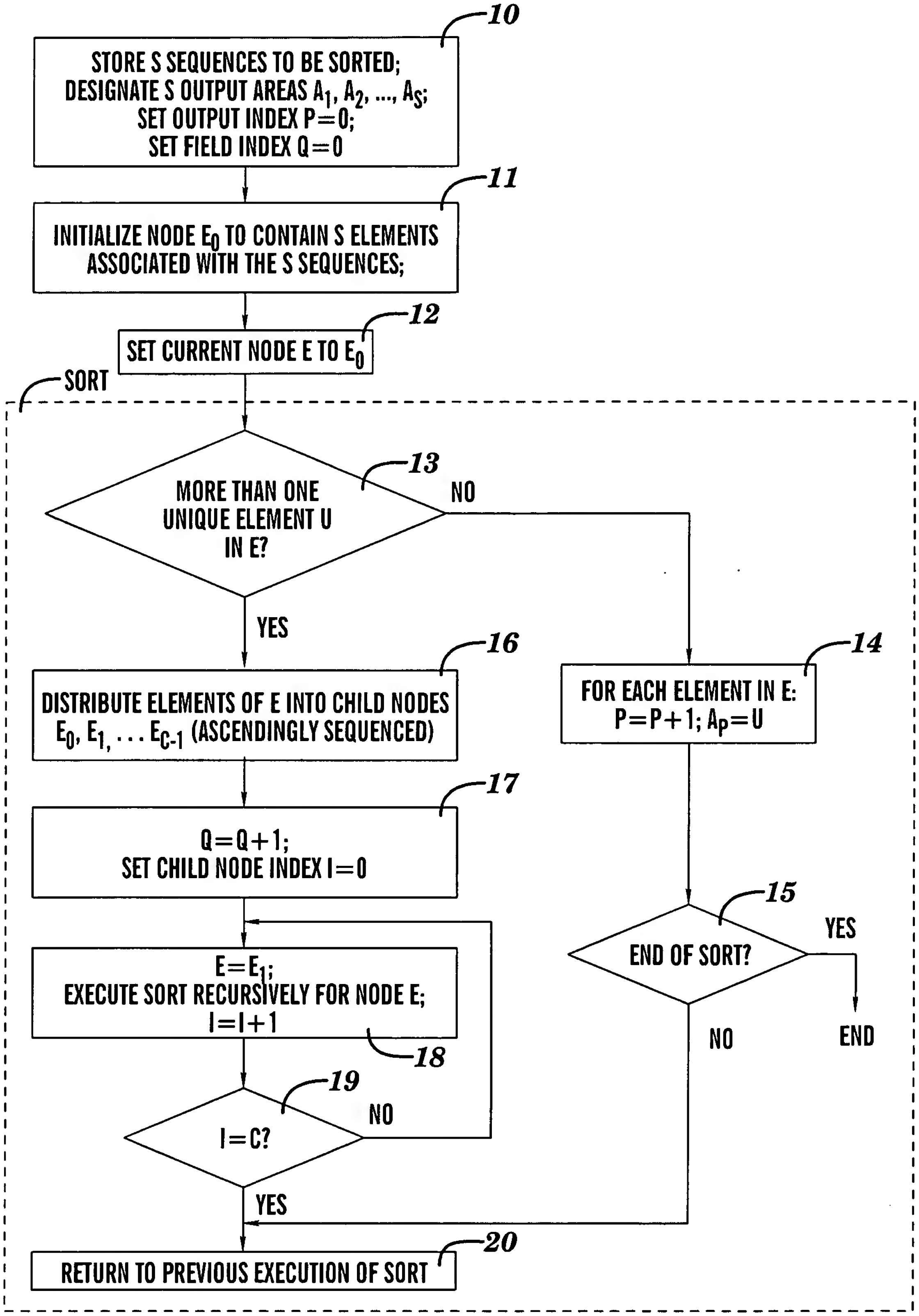

10
STORE S SEQUENCES TO BE SORTED;
DESIGNATE S OUTPUT AREAS A_1, A_2, ..., A_S;
SET OUTPUT INDEX P=0;
SET FIELD INDEX Q=0
11
INITIALIZE NODE E_0 TO CONTAIN S ELEMENTS ASSOCIATED WITH THE S SEQUENCES;
12
SET CURRENT NODE E TO E_0
SORT
13
MORE THAN ONE UNIQUE ELEMENT U IN E?
NO
YES
16
DISTRIBUTE ELEMENTS OF E INTO CHILD NODES E_0, E_1, . . . E_{C-1} (ASCENDINGLY SEQUENCED)
14
FOR EACH ELEMENT IN E:
P=P+1; A_P=U
17
Q=Q+1;
SET CHILD NODE INDEX I=0
15
END OF SORT?
YES
END
NO
E=E_I;
EXECUTE SORT RECURSIVELY FOR NODE E;
I=I+1
18
19
I=C?
NO
YES
20
RETURN TO PREVIOUS EXECUTION OF SORT

FIG. 5
(RECURSIVE EXECUTION)

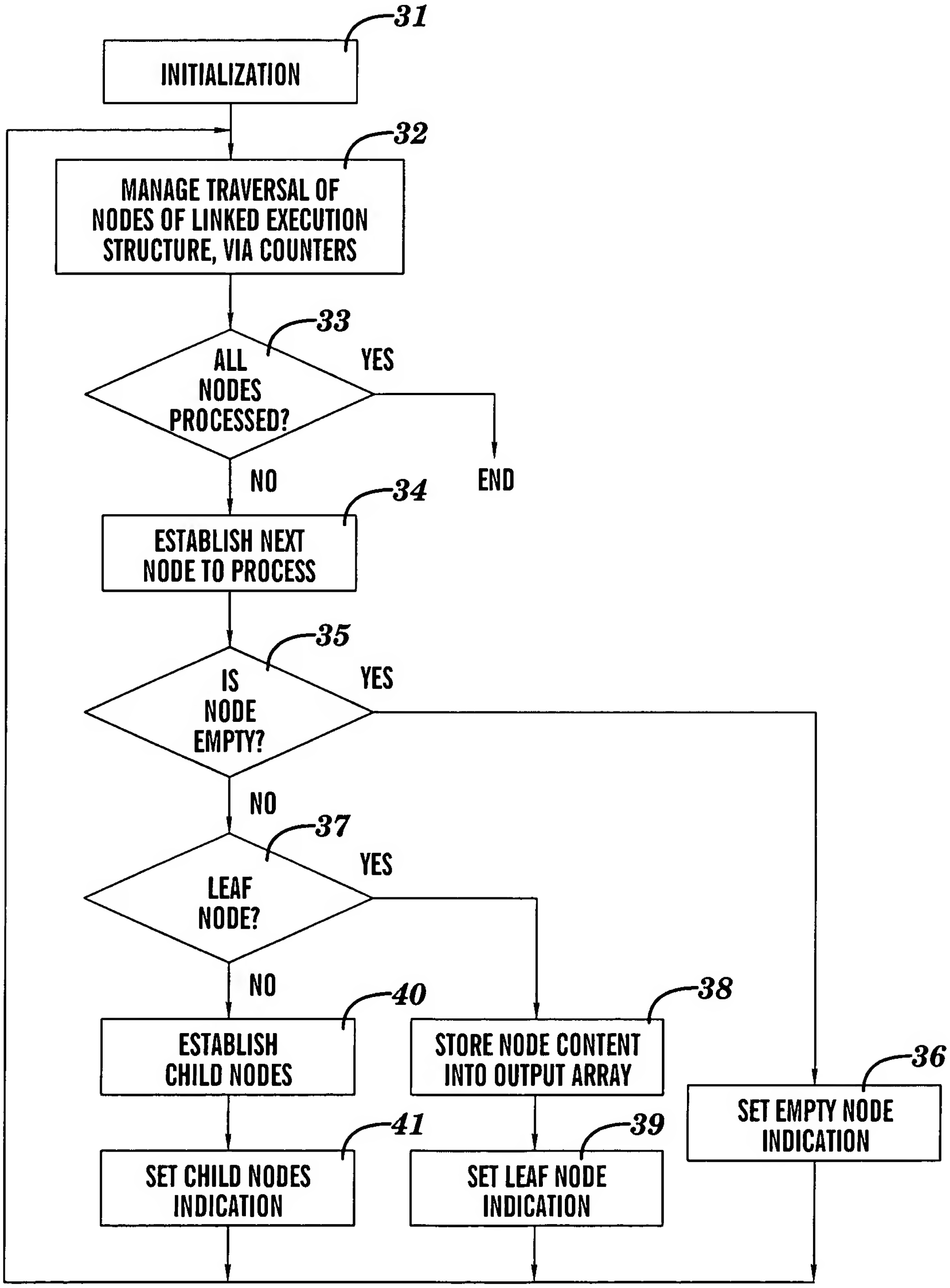
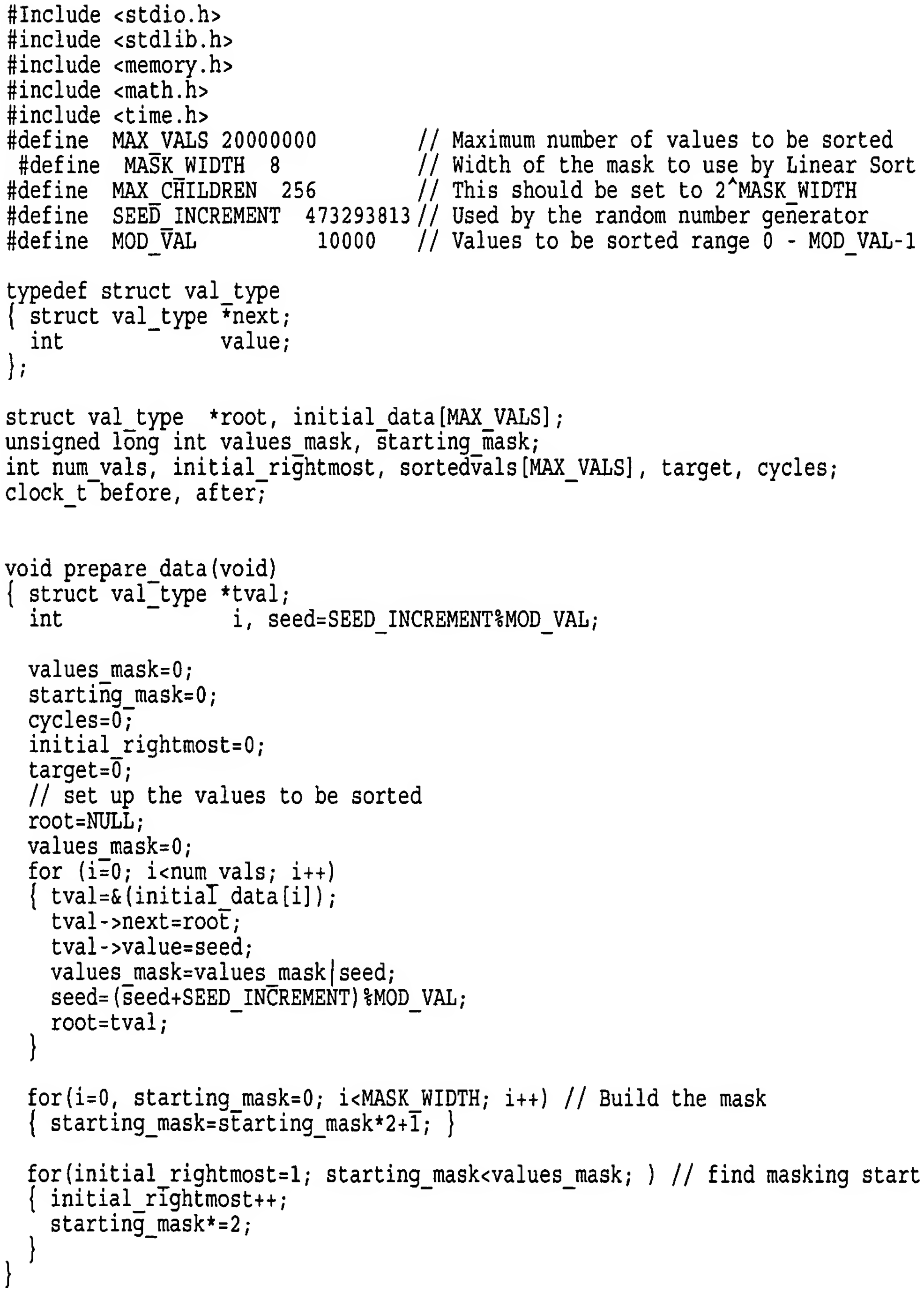

31
INITIALIZATION
32
MANAGE TRAVERSAL OF NODES OF LINKED EXECUTION STRUCTURE, VIA COUNTERS
33
ALL NODES PROCESSED?
YES
END
NO
34
ESTABLISH NEXT NODE TO PROCESS
35
IS NODE EMPTY?
YES
NO
37
LEAF NODE?
YES
NO
40
ESTABLISH CHILD NODES
38
STORE NODE CONTENT INTO OUTPUT ARRAY
36
SET EMPTY NODE INDICATION
41
SET CHILD NODES INDICATION
39
SET LEAF NODE INDICATION

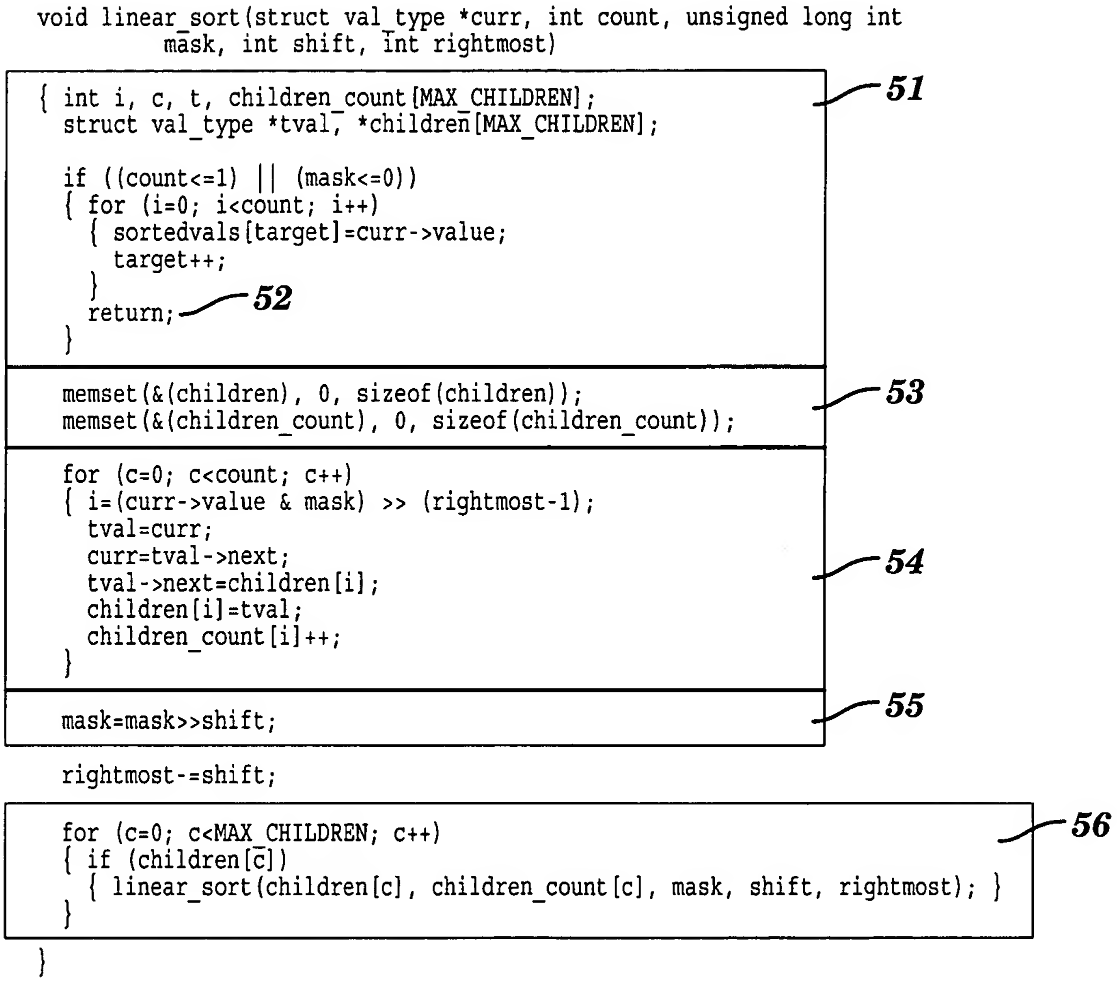
FIG. 6

(COUNTER-CONTROLLED LOOPING)



```
#Include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <math.h>
#include <time.h>
#define  MAX_VALS 20000000        // Maximum number of values to be sorted
 #define  MASK_WIDTH  8           // Width of the mask to use by Linear Sort
#define  MAX_CHILDREN  256        // This should be set to 2^MASK_WIDTH
#define  SEED_INCREMENT  473293813// Used by the random number generator
#define  MOD_VAL          10000   // Values to be sorted range 0 - MOD_VAL-1

typedef struct val_type
{ struct val_type *next;
  int             value;
};

struct val_type  *root, initial_data[MAX_VALS];
unsigned long int values_mask, starting_mask;
int num_vals, initial_rightmost, sortedvals[MAX_VALS], target, cycles;
clock_t before, after;


void prepare_data(void)
{ struct val_type *tval;
  int              i, seed=SEED_INCREMENT%MOD_VAL;

  values_mask=0;
  starting_mask=0;
  cycles=0;
  initial_rightmost=0;
  target=0;
  // set up the values to be sorted
  root=NULL;
  values_mask=0;
  for (i=0; i<num_vals; i++)
  { tval=&(initial_data[i]);
    tval->next=root;
    tval->value=seed;
    values_mask=values_mask|seed;
    seed=(seed+SEED_INCREMENT)%MOD_VAL;
    root=tval;
  }

  for(i=0, starting_mask=0; i<MASK_WIDTH; i++) // Build the mask
  { starting_mask=starting_mask*2+1; }

  for(initial_rightmost=1; starting_mask<values_mask; ) // find masking start
  { initial_rightmost++;
    starting_mask*=2;
  }
}
```

FIG. 7A



```
void linear_sort(struct val_type *curr, int count, unsigned long int
          mask, int shift, int rightmost)
{ int i, c, t, children_count[MAX_CHILDREN];
  struct val_type *tval, *children[MAX_CHILDREN];

  if ((count<=1) || (mask<=0))
  { for (i=0; i<count; i++)
    { sortedvals[target]=curr->value;
      target++;
    }
    return;
  }
  memset(&(children), 0, sizeof(children));
  memset(&(children_count), 0, sizeof(children_count));
  for (c=0; c<count; c++)
  { i=(curr->value & mask) >> (rightmost-1);
    tval=curr;
    curr=tval->next;
    tval->next=children[i];
    children[i]=tval;
    children_count[i]++;
  }
  mask=mask>>shift;
  rightmost-=shift;
  for (c=0; c<MAX_CHILDREN; c++)
  { if (children[c])
    { linear_sort(children[c], children_count[c], mask, shift, rightmost); }
  }
}
```

FIG. 7B

```
void quicksort(int lo0, int hi0)
{ int lo = lo0;
  int hi = hi0;
  int pivot, t;

  if (lo >= hi) { return; }
  else if( lo == hi - 1 )
  { if (sortedvals[lo] > sortedvals[hi])
    { t = sortedvals[lo];
      sortedvals[lo] = sortedvals[hi];
      sortedvals[hi] = t;
    }
    return;
  }

  pivot = sortedvals[(lo + hi) / 2];
  sortedvals[(lo + hi) / 2] = sortedvals[hi];
  sortedvals[hi] = pivot;

  while( lo < hi )
  { while ((sortedvals[lo] <= pivot) && (lo < hi))
    { lo++; }

    while ((pivot <= sortedvals[hi]) && (lo < hi ))
    { hi--; }

    if (lo < hi)
    { t              = sortedvals[lo];
      sortedvals[lo] = sortedvals[hi];
      sortedvals[hi] = t;
    }
  }

  sortedvals[hi0] = sortedvals[hi];
  sortedvals[hi] = pivot;
  quicksort(lo0, lo-1);
  quicksort(hi+1, hi0);
}
```

FIG. 7C

```
void main(void)
{
  printf("#_Values\t\tLinear\t\t\tQuicksort\n");

  for (num_vals=1000000; num_vals<=MAX_VALS; num_vals+=1000000)
  { prepare_data();
    before=clock();
    linear_sort(root, num_vals, starting_mask, MASK_WIDTH, initial_rightmost);
    after=clock();
    printf("%10d\t%10d\t%10d\t", num_vals, cycles, after-before);

    build_dataset();
    before=clock();
    quicksort(0, num_vals);
    after=clock();
    printf("%10l\t%10d", cycles, after-before);
    printf("\n");
  }
}

void build_dataset(void)
{ int i, high, low, avg, counts[MOD_VAL];

  cycles=0;
  sortedvals[0]=SEED_INCREMENT%MOD_VAL;

  for (i=1; i<num_vals; i++)
  { sortedvals[i] = (sortedvals[i-1]+SEED_INCREMENT)%MOD_VAL; }
}
```

FIG. 7D

The following source code sample contains both the Linear Sort and the Quicksort Algorithms.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <memory.h>
#include <time.h>
#define  MAX_VALS 1000000          // Maximum number of values to be sorted
#define  TEST_INCREMENT 10000      // Maximum number of values to be sorted
#define  MAX_STR_LEN   20          // Maximum length of strings to be sorted
#define  MAX_CHILDREN 256          // 256 because the Mask Width here is 8bits

typedef struct val_type
{ struct val_type *next;
  char            *value;
};
struct val_type  *root, initial_data[MAX_VALS];
long num_vals, target, cycles, moves, compares;
char *sortedvals[MAX_VALS], raw_data[MAX_VALS][MAX_STR_LEN];
clock_t before, after;
FILE *infile;

void prepare_data(void)
{ struct val_type *tval;
  int              i;

  target=0;
  // set up the values to be sorted
  root=NULL;
  for (i=0; i<num_vals; i++)
  { tval=&(initial_data[i]);
    tval->next=root;
    tval->value=&(raw_data[i][0]);
    root=tval;
  }
}
```

FIG. 8A

```
void linear_sort(struct val_type *curr, int count, int level)
{ int i, c, t, children_count[MAX_CHILDREN];
  struct val_type *tval, *children[MAX_CHILDREN];

  if (count==1)
  { sortedvals[target]=curr->value;
    target++;
    return;
  }
  memset(&(children), 0, sizeof(children));
  memset(&(children_count), 0, sizeof(children_count));

  for (c=0; c<count; c++)                                      60
  { i=curr->value[level];
    cycles++;
    if (i==0)
    { sortedvals[target]=curr->value;
      target++;
    }
    else
    { tval=curr;
      curr=curr->next;
      tval->next=children[i];
      children[i]=tval;
      children_count[i]++;
    }
  }

  for (c=1; c<MAX_CHILDREN; c++)
  { if (children[c])
    { linear_sort(children[c], children_count[c], level+1); }
  }
}


void validate_sort(void)
{ int i;

  for (i=1; i<num_vals; i++)
  { if (strcmp(sortedvals[i-1],sortedvals[i])>0)
    { printf("sort error=> %d:[%s][%s]\n", i, sortedvals[i-1],sortedvals[i]);
      return;
    }
  }
  printf(" OK ");
}
```

FIG. 8B

```
void quicksort(int lo0, int hi0)
{ int lo = lo0;
  int hi = hi0;
  char *pivot, *t;

  if (lo >= hi) { return; }
  else if( lo == hi - 1 )
  { if (strcmp(sortedvals[lo], sortedvals[hi])>0)
    { t = sortedvals[lo];
      sortedvals[lo] = sortedvals[hi];
      sortedvals[hi] = t;
    }
    compares++;
    return;
  }

  pivot = sortedvals[(lo + hi) / 2];
  sortedvals[(lo + hi) / 2] = sortedvals[hi];
  sortedvals[hi] = pivot;

  while( lo < hi )
  { while ((strcmp(sortedvals[lo],pivot)<=0) && (lo < hi))
    { lo++;
      compares++;
    }
    compares++;

    while ((strcmp(pivot,sortedvals[hi])<=0) && (lo < hi ))
    { hi--;
      compares++;
    }
    compares++;

    if (lo < hi)
    { t              = sortedvals[lo];
      sortedvals[lo] = sortedvals[hi];
      sortedvals[hi] = t;
      moves++;
    }
  }

  sortedvals[hi0] = sortedvals[hi];
  sortedvals[hi] = pivot;
  quicksort(lo0, lo-1);
  quicksort(hi+1, hi0);
}
```

FIG. 8C

```
void build_dataset(void)
{ int i, c=0, m=0, p=0;

  infile=fopen("strings.dat", "r");
  for (i=0; i<MAX_VALS; i++)
  { fscanf(infile, "%s\n", &(raw_data[i]));
    if (strlen(raw_data[i])>m)
    { m=strlen(raw_data[i]);
      p=i;
    }
  }
  fclose(infile);
  printf("max string length=%d at %d\n", m, p);
}

void reset_dataset(void)
{ int i;

  for (i=0; i<num_vals; i++)
  { sortedvals[i]=&(raw_data[i][0]); }
}

void dump_dataset(void)
{ int i;

  for (i=0; i<MAX_VALS; i++)
  { printf("%d: %s\n", i, raw_data[i]); }
  for (i=0; i<MAX_VALS; i++)
  { printf("%d: %s\n", i, sortedvals[i]); }
}

void main(void)
{
  build_dataset();
  printf("\t\tQuicksort\t\t\tLinear\n");
  printf("#_Values   compares      moves    clock         cycles   clock\n");

  for (num_vals=TEST_INCREMENT; num_vals<=MAX_VALS; num_vals+=TEST_INCREMENT)
  { reset_dataset();
    compares=0;
    moves=0;
    printf("%10d ", num_vals);
    before=clock();
    quicksort(0, num_vals-1);
    after=clock();
    printf("%10d %10d %6d", compares, moves, after-before);

    cycles=0;
    prepare_data();
    reset_dataset();
    before=clock();
    linear_sort(root, num_vals, 0);
    after=clock();
    printf("   %10d %6d", cycles, after-before);
    printf("\n");
  }
}
```

FIG. 8D

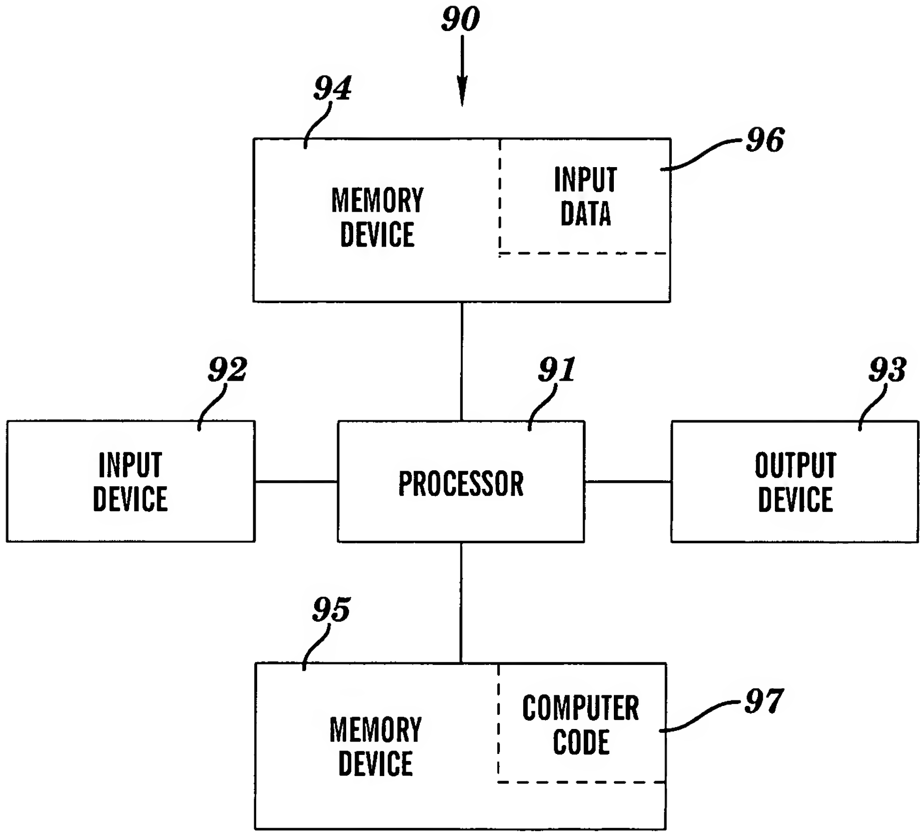

90
94
MEMORY DEVICE
INPUT DATA
96
92
INPUT DEVICE
91
PROCESSOR
93
OUTPUT DEVICE
95
MEMORY DEVICE
COMPUTER CODE
97

FIG. 9

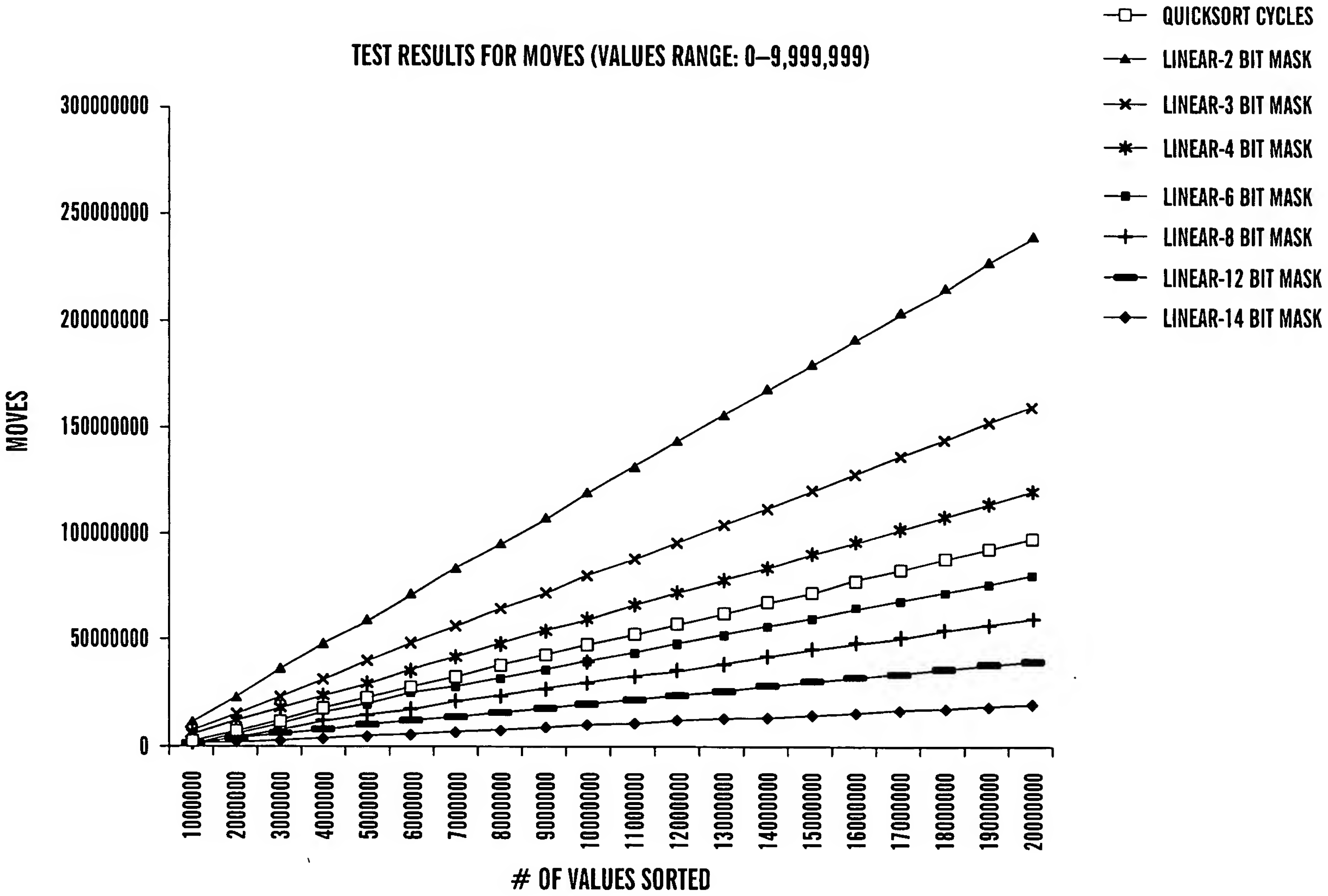

TEST RESULTS FOR MOVES (VALUES RANGE: 0–9,999,999)
MOVES
300000000
250000000
200000000
150000000
100000000
50000000
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000
9000000
10000000
11000000
12000000
13000000
14000000
15000000
16000000
17000000
18000000
19000000
20000000
# OF VALUES SORTED
QUICKSORT CYCLES
LINEAR-2 BIT MASK
LINEAR-3 BIT MASK
LINEAR-4 BIT MASK
LINEAR-6 BIT MASK
LINEAR-8 BIT MASK
LINEAR-12 BIT MASK
LINEAR-14 BIT MASK

FIG. 10

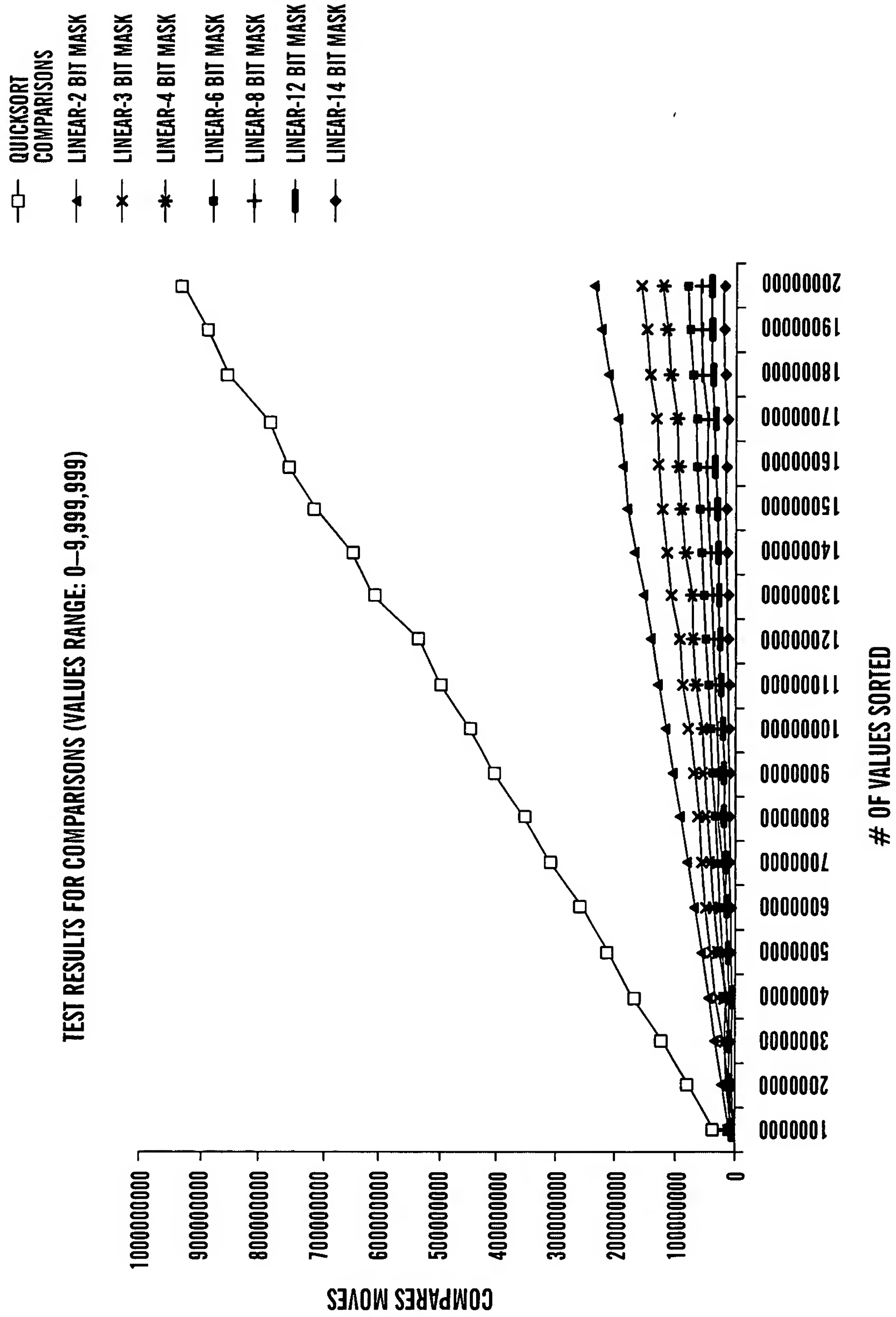

TEST RESULTS FOR COMPARISONS (VALUES RANGE: 0–9,999,999)
COMPARES MOVES
1000000000
900000000
800000000
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000
9000000
10000000
11000000
12000000
13000000
14000000
15000000
16000000
17000000
18000000
19000000
20000000
# OF VALUES SORTED
QUICKSORT COMPARISONS
LINEAR-2 BIT MASK
LINEAR-3 BIT MASK
LINEAR-4 BIT MASK
LINEAR-6 BIT MASK
LINEAR-8 BIT MASK
LINEAR-12 BIT MASK
LINEAR-14 BIT MASK

FIG. 11

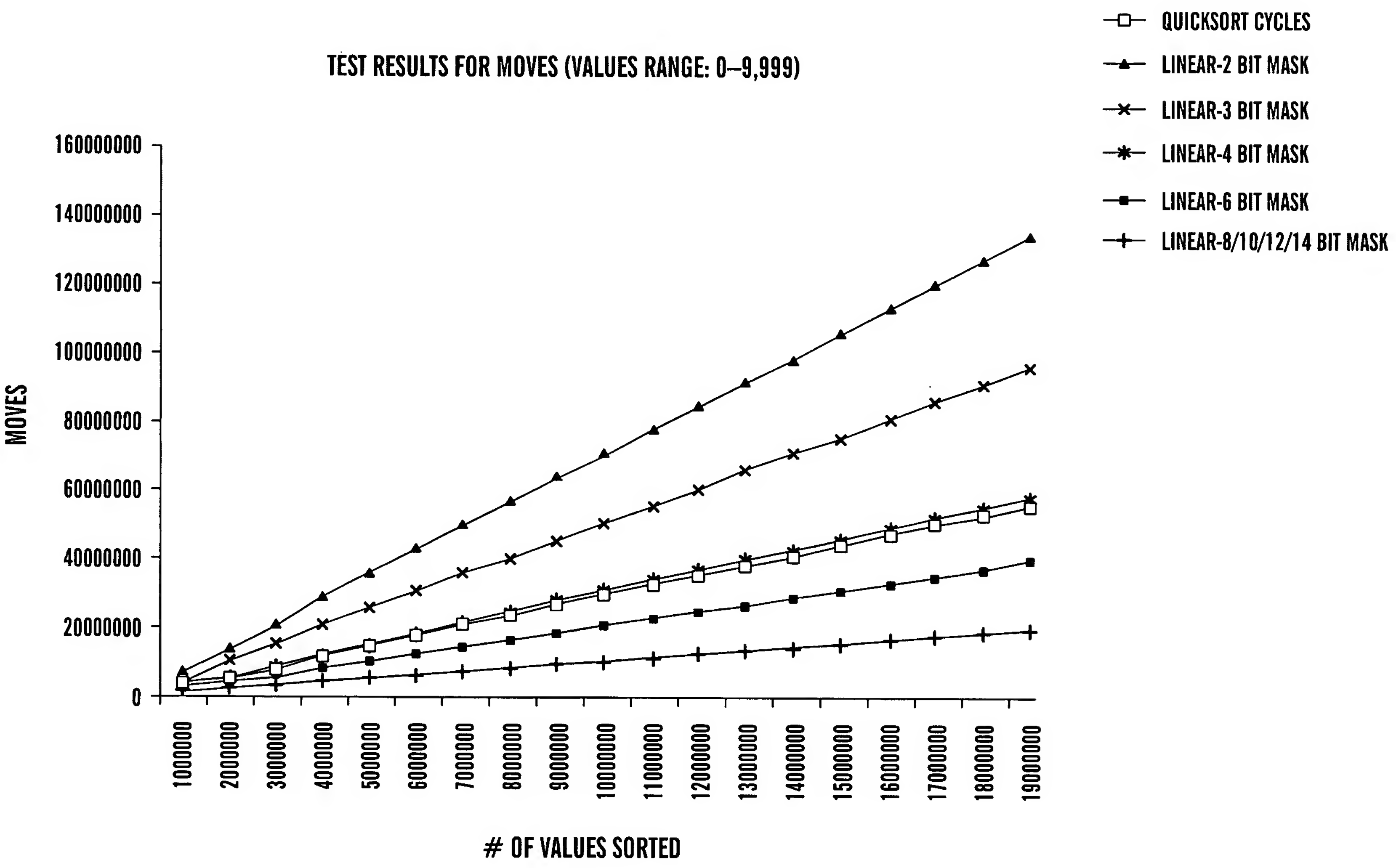

TEST RESULTS FOR MOVES (VALUES RANGE: 0–9,999)
MOVES
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000
9000000
10000000
11000000
12000000
13000000
14000000
15000000
16000000
17000000
18000000
19000000
# OF VALUES SORTED
QUICKSORT CYCLES
LINEAR-2 BIT MASK
LINEAR-3 BIT MASK
LINEAR-4 BIT MASK
LINEAR-6 BIT MASK
LINEAR-8/10/12/14 BIT MASK

FIG. 12

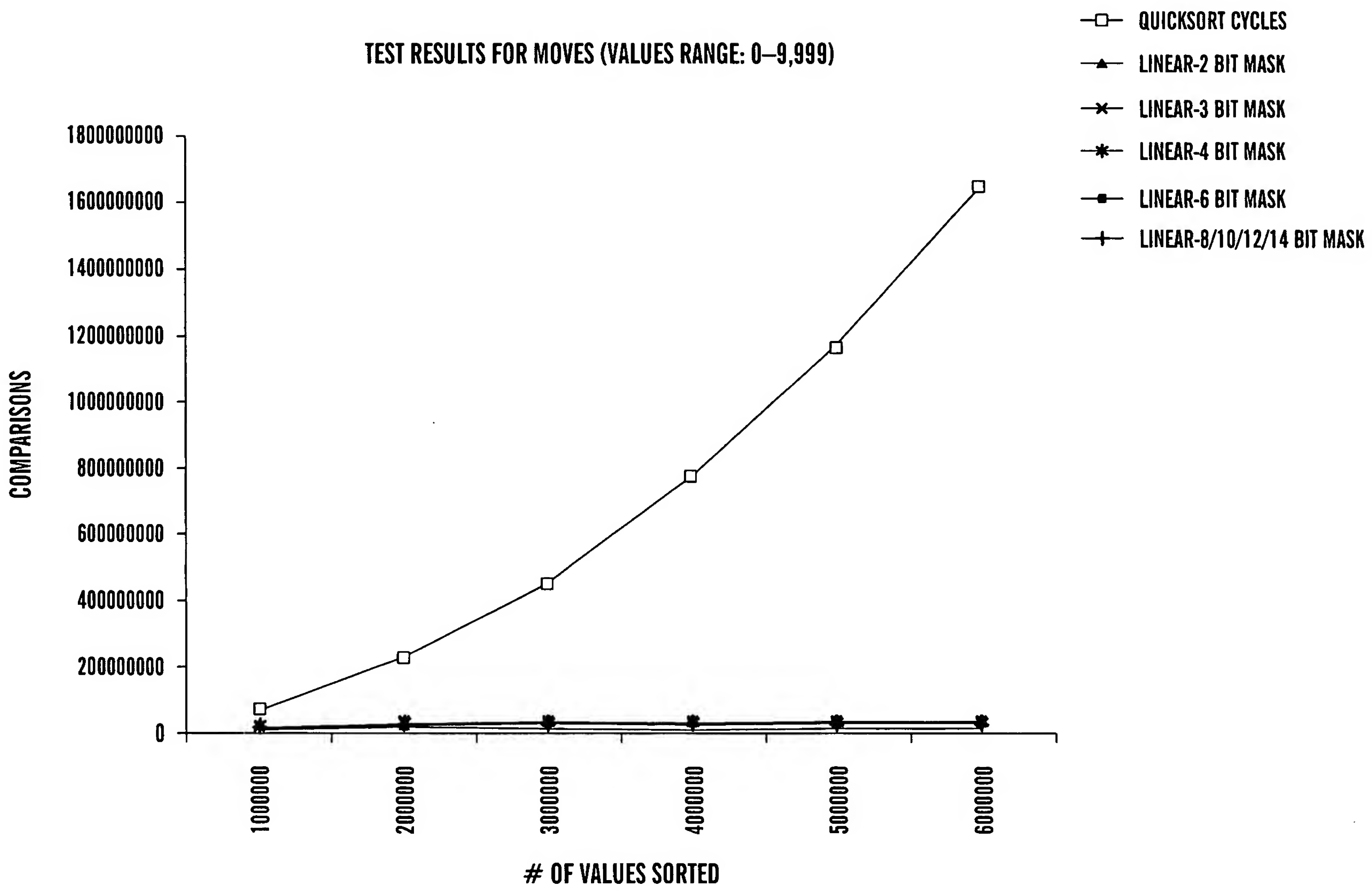
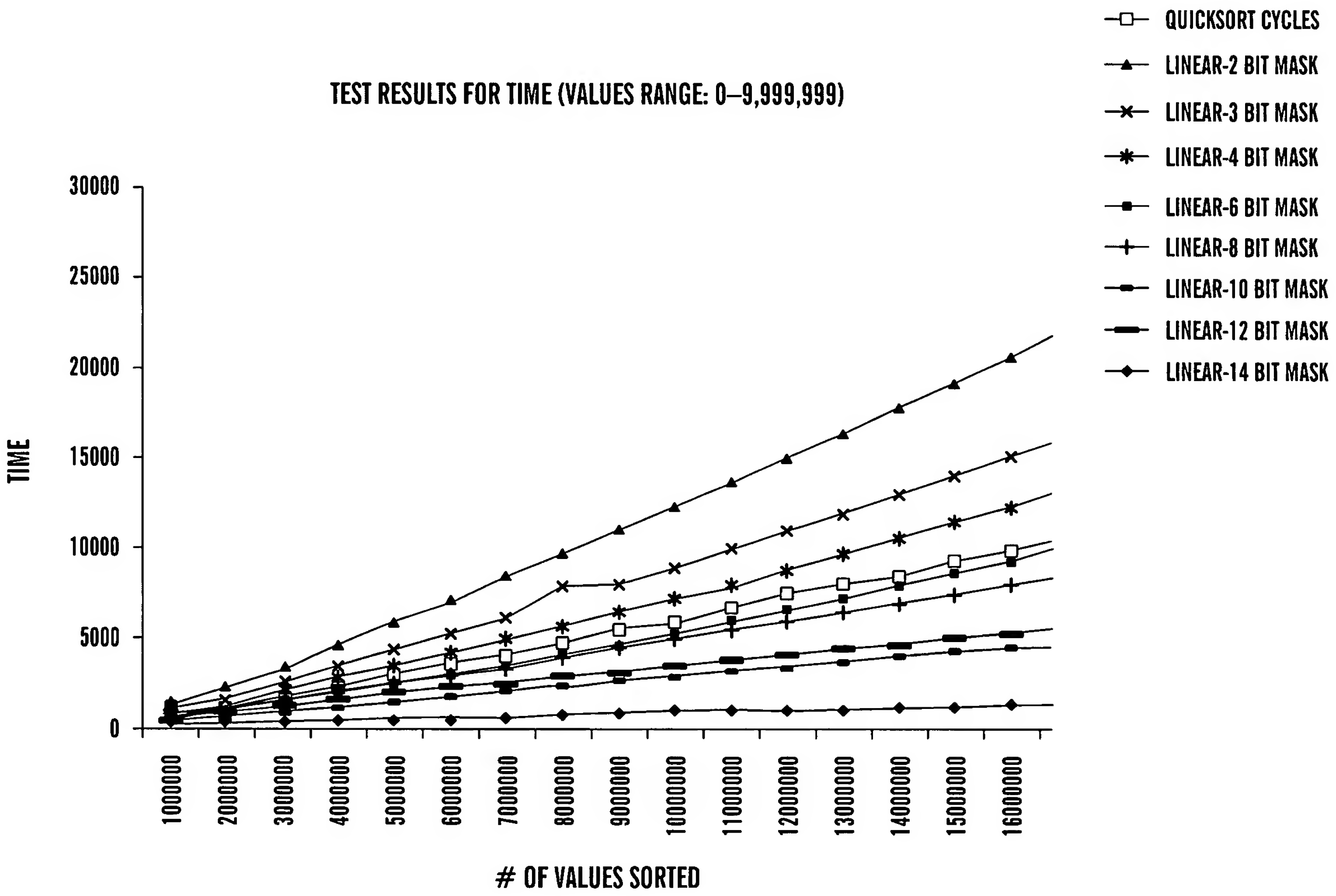

TEST RESULTS FOR MOVES (VALUES RANGE: 0–9,999)
QUICKSORT CYCLES
LINEAR-2 BIT MASK
LINEAR-3 BIT MASK
LINEAR-4 BIT MASK
LINEAR-6 BIT MASK
LINEAR-8/10/12/14 BIT MASK
COMPARISONS
1800000000
1600000000
1400000000
1200000000
1000000000
800000000
600000000
400000000
200000000
0
1000000
2000000
3000000
4000000
5000000
6000000
# OF VALUES SORTED

FIG. 13



TEST RESULTS FOR TIME (VALUES RANGE: 0–9,999,999)

FIG. 14

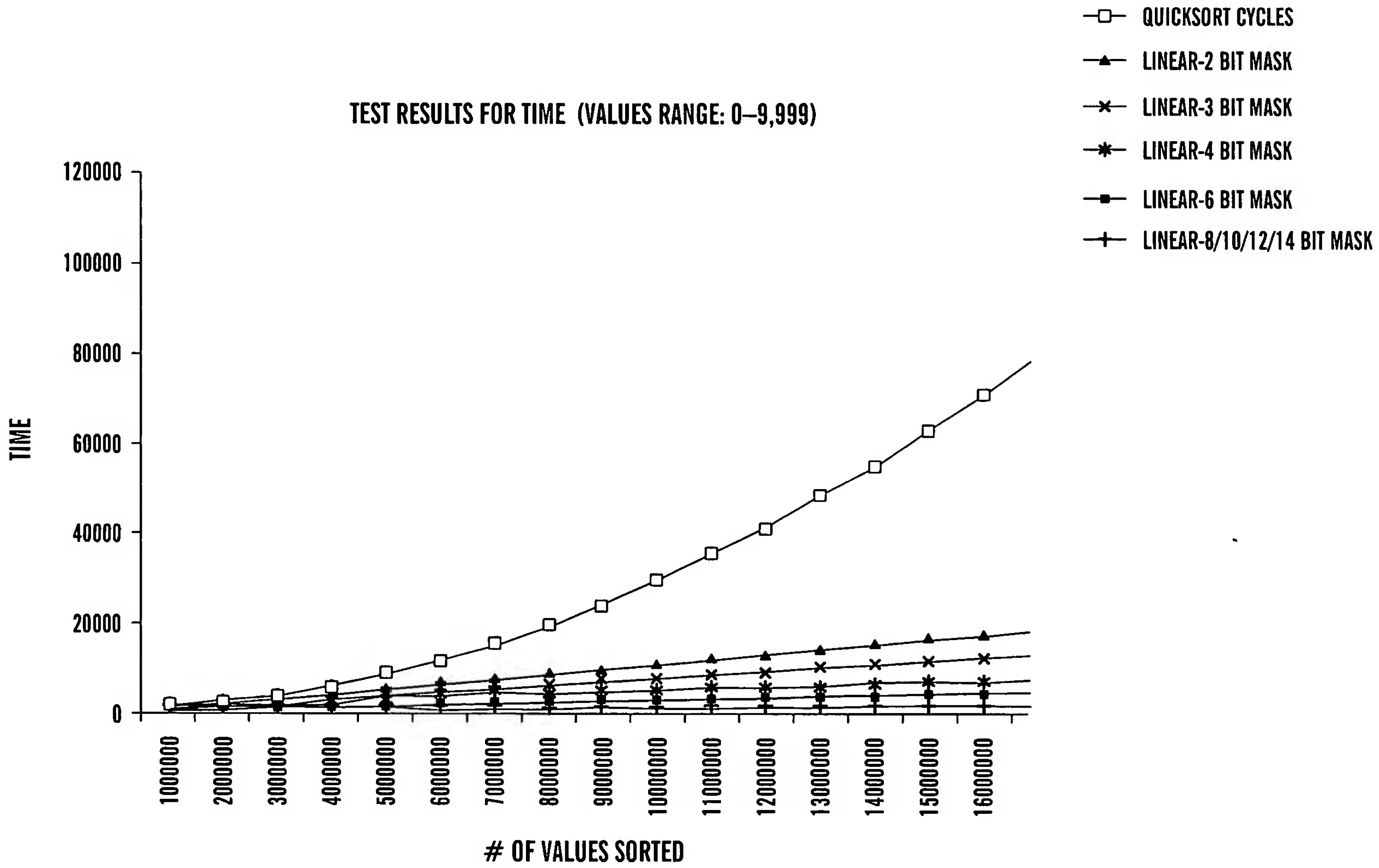

TEST RESULTS FOR TIME (VALUES RANGE: 0–9,999)
TIME
120000
100000
80000
60000
40000
20000
0
1000000
2000000
3000000
4000000
5000000
6000000
7000000
8000000
9000000
10000000
11000000
12000000
13000000
14000000
15000000
16000000
# OF VALUES SORTED
QUICKSORT CYCLES
LINEAR-2 BIT MASK
LINEAR-3 BIT MASK
LINEAR-4 BIT MASK
LINEAR-6 BIT MASK
LINEAR-8/10/12/14 BIT MASK

FIG. 15

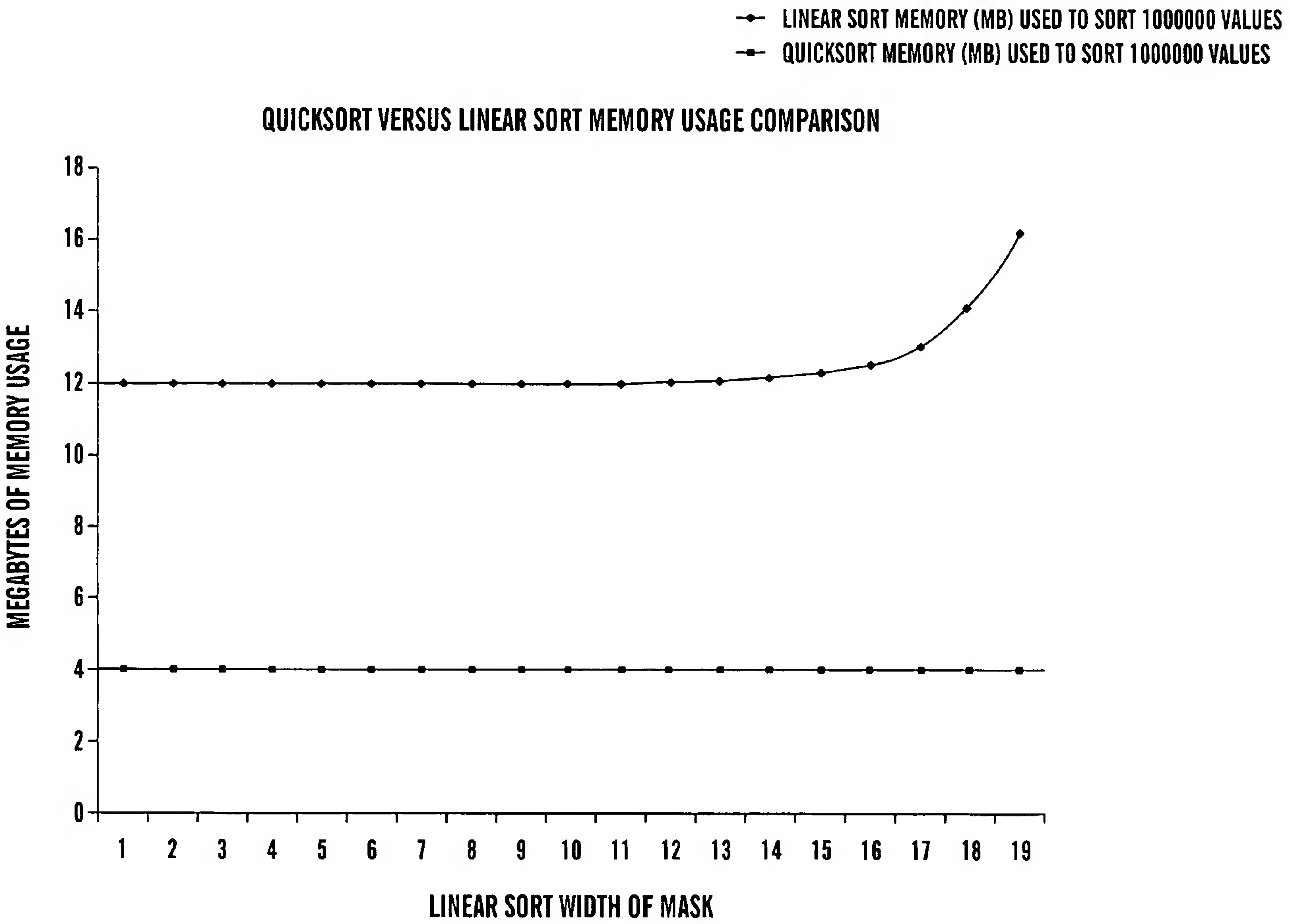

LINEAR SORT MEMORY (MB) USED TO SORT 1000000 VALUES
QUICKSORT MEMORY (MB) USED TO SORT 1000000 VALUES
QUICKSORT VERSUS LINEAR SORT MEMORY USAGE COMPARISON
MEGABYTES OF MEMORY USAGE
18
16
14
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
LINEAR SORT WIDTH OF MASK

FIG. 16

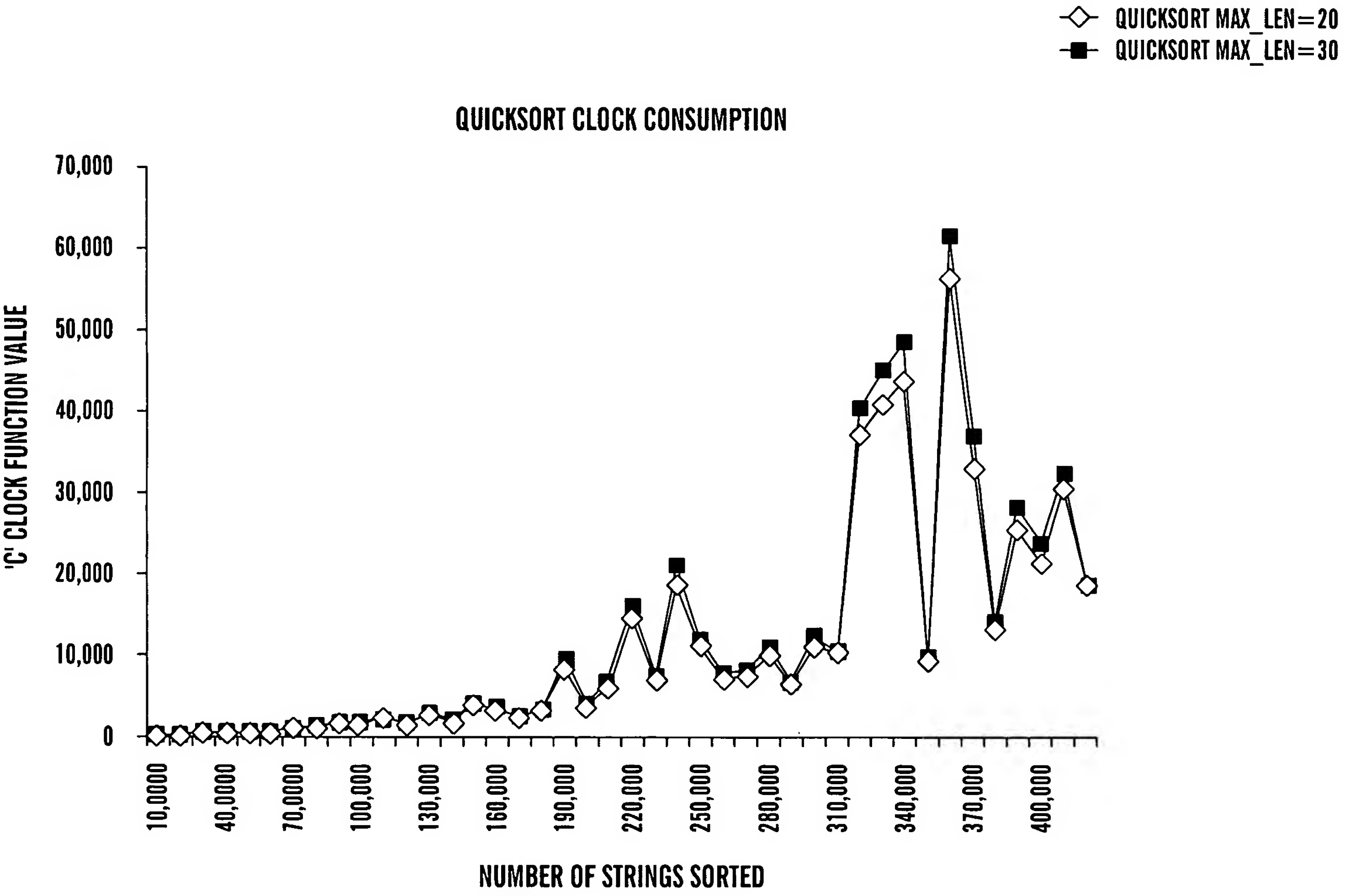

QUICKSORT MAX_LEN=20
QUICKSORT MAX_LEN=30
QUICKSORT CLOCK CONSUMPTION
'C' CLOCK FUNCTION VALUE
70,000
60,000
50,000
40,000
30,000
20,000
10,000
0
10,0000
40,0000
70,0000
100,000
130,000
160,000
190,000
220,000
250,000
280,000
310,000
340,000
370,000
400,000
NUMBER OF STRINGS SORTED

FIG. 17

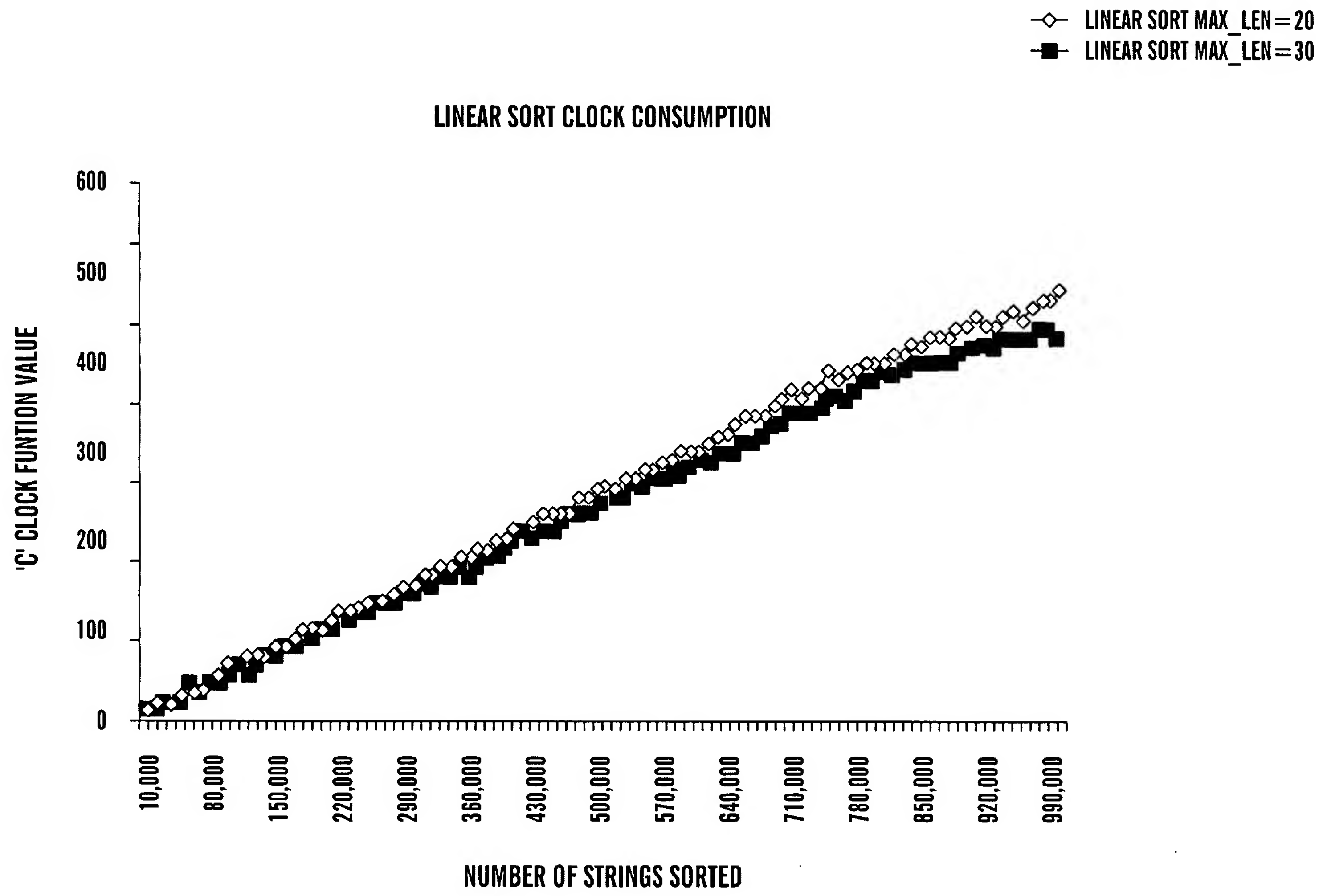

LINEAR SORT MAX_LEN=20
LINEAR SORT MAX_LEN=30
LINEAR SORT CLOCK CONSUMPTION
'C' CLOCK FUNTION VALUE
600
500
400
300
200
100
0
10,000
80,000
150,000
220,000
290,000
360,000
430,000
500,000
570,000
640,000
710,000
780,000
850,000
920,000
990,000
NUMBER OF STRINGS SORTED

FIG. 18

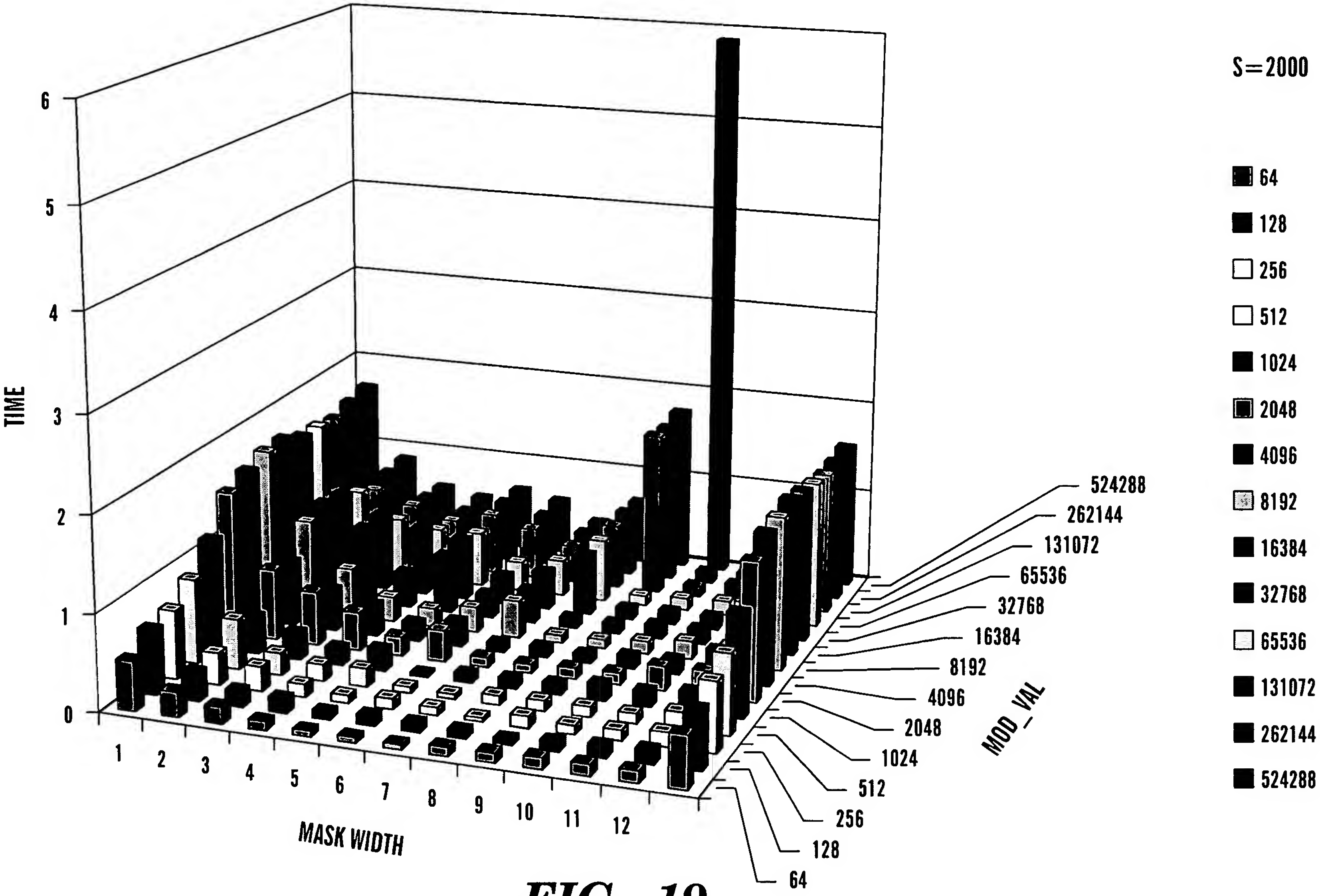

TIME
0
1
2
3
4
5
6
1
2
3
4
5
6
7
8
9
10
11
12
MASK WIDTH
MOD_VAL
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
524288
S=2000
▩ 64
■ 128
□ 256
□ 512
■ 1024
▩ 2048
■ 4096
▨ 8192
■ 16384
■ 32768
□ 65536
■ 131072
■ 262144
■ 524288

FIG. 19

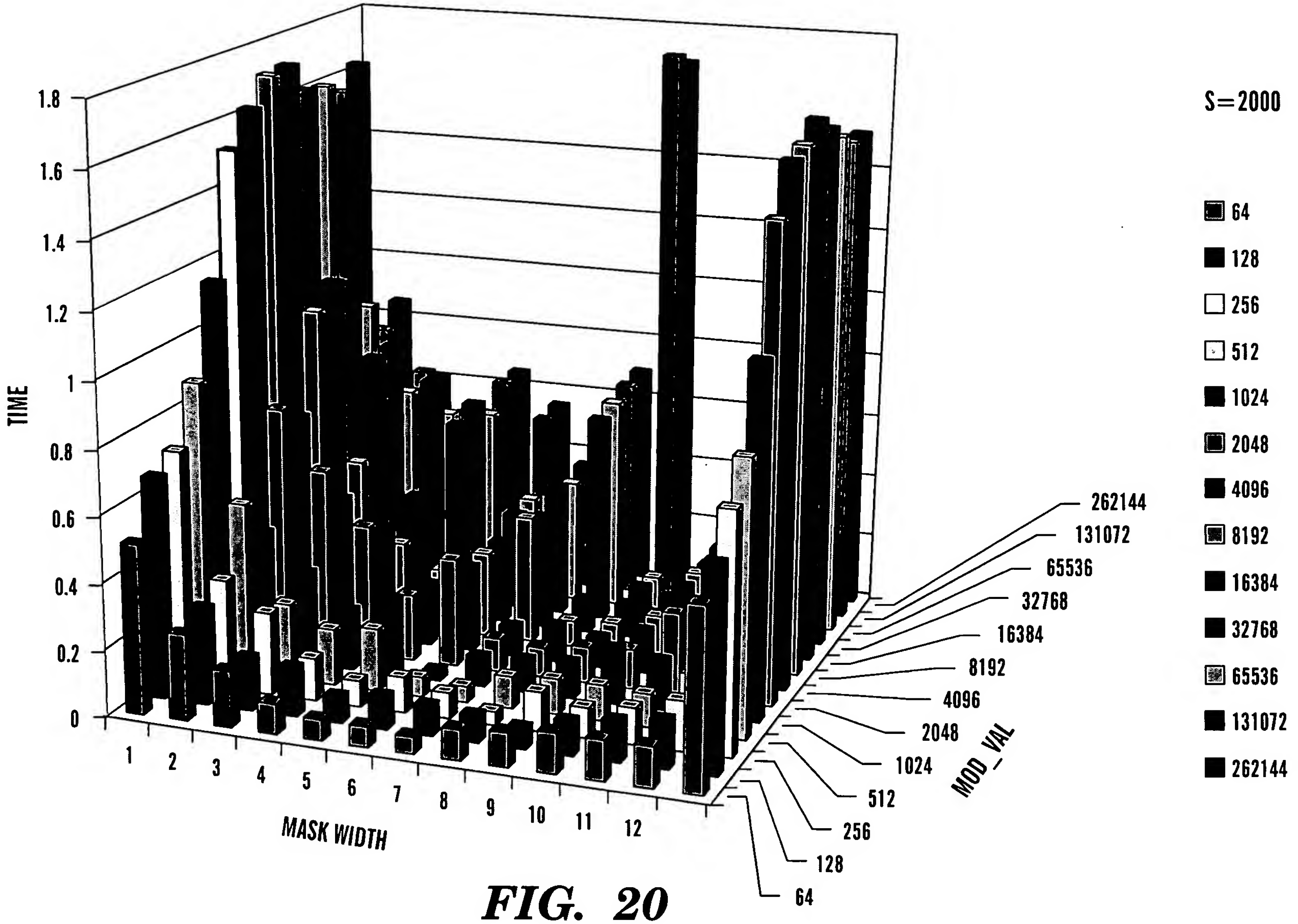

TIME
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10
11
12
MASK WIDTH
MOD_VAL
262144
131072
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
S=2000
▨ 64
■ 128
□ 256
□ 512
■ 1024
▨ 2048
■ 4096
▨ 8192
■ 16384
■ 32768
▨ 65536
■ 131072
■ 262144

FIG. 20

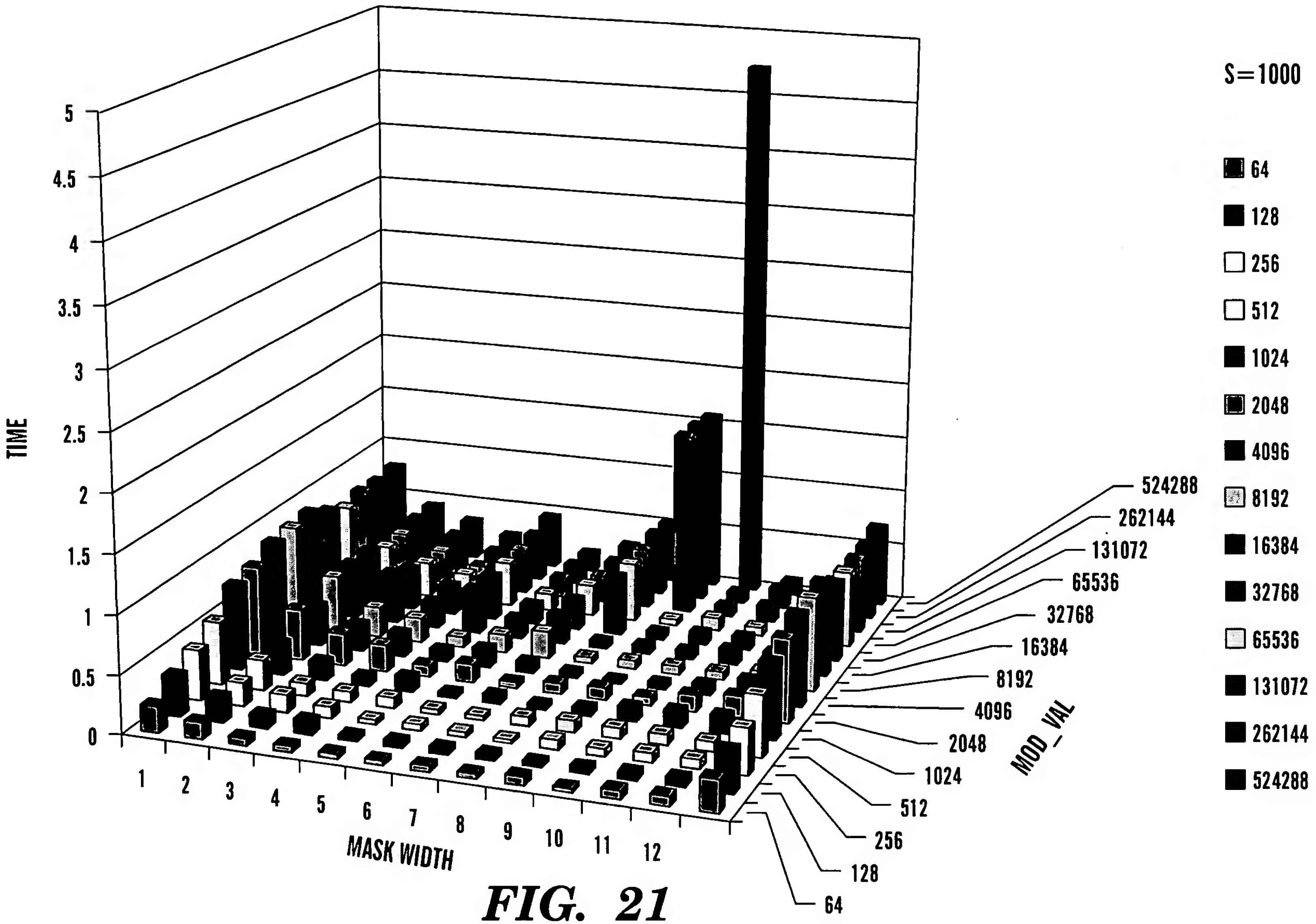

S=1000
TIME
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
9
10
11
12
MASK WIDTH
MOD_VAL
524288
262144
131072
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
■ 64
■ 128
□ 256
□ 512
■ 1024
■ 2048
■ 4096
▨ 8192
■ 16384
■ 32768
▨ 65536
■ 131072
■ 262144
■ 524288

FIG. 21

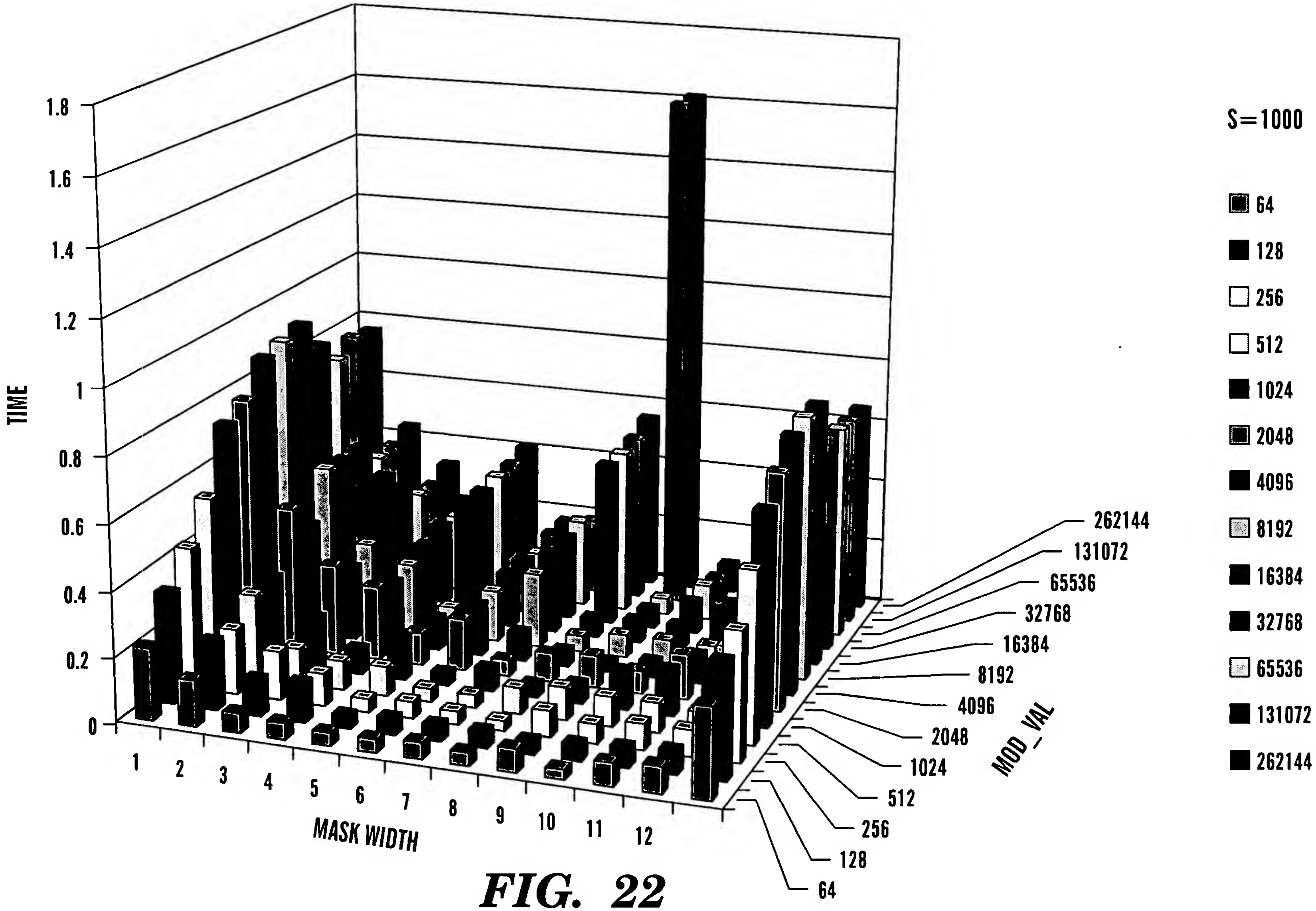

TIME
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
MASK WIDTH
1
2
3
4
5
6
7
8
9
10
11
12
MOD_VAL
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
S=1000
▣ 64
■ 128
□ 256
□ 512
■ 1024
▣ 2048
■ 4096
▨ 8192
■ 16384
■ 32768
▨ 65536
■ 131072
■ 262144

FIG. 22

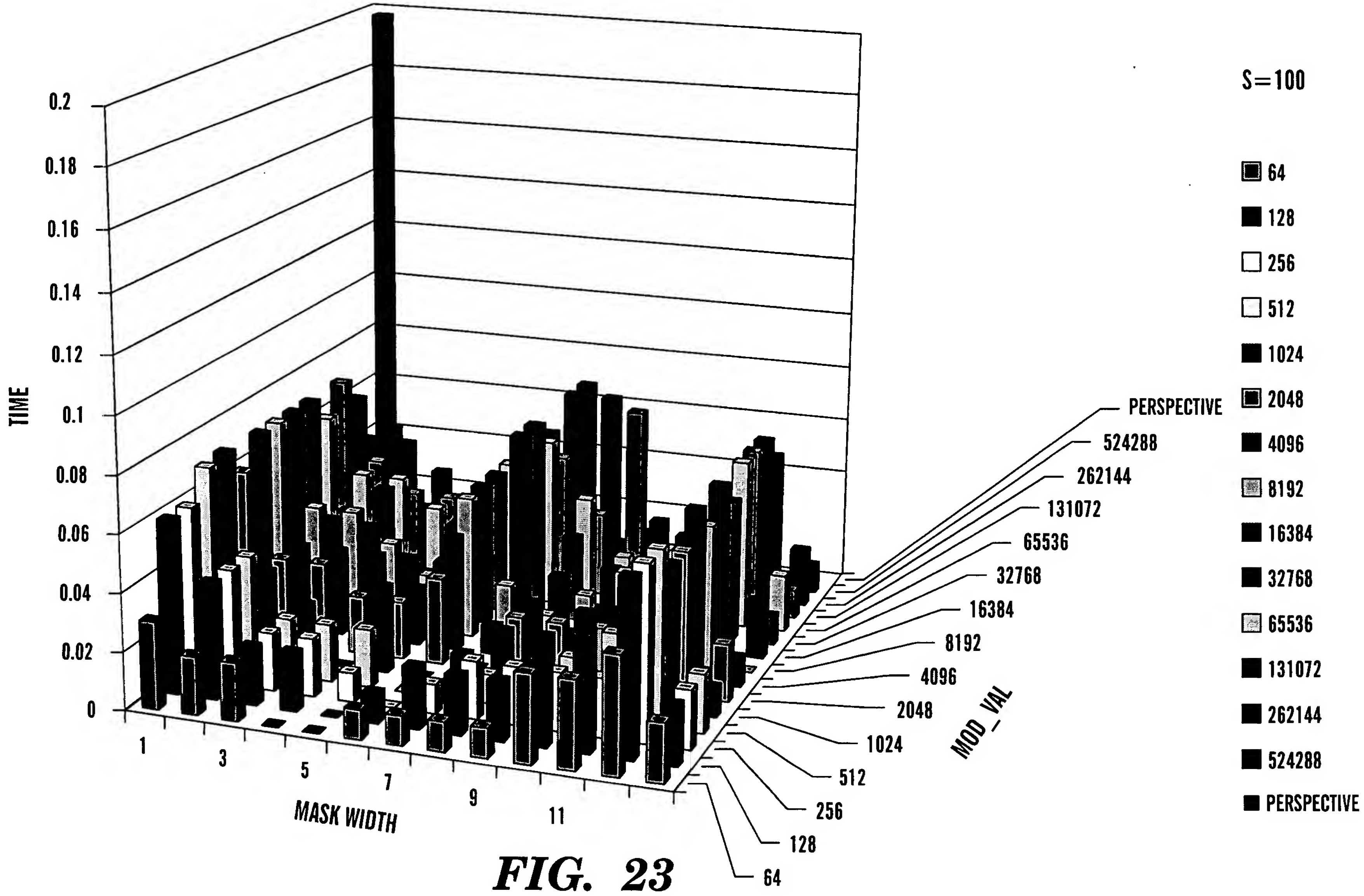

TIME
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
1
3
5
7
9
11
MASK WIDTH
MOD_VAL
PERSPECTIVE
524288
262144
131072
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
S=100
▣ 64
■ 128
□ 256
□ 512
■ 1024
▣ 2048
■ 4096
▨ 8192
■ 16384
■ 32768
▨ 65536
■ 131072
■ 262144
■ 524288
■ PERSPECTIVE

FIG. 23

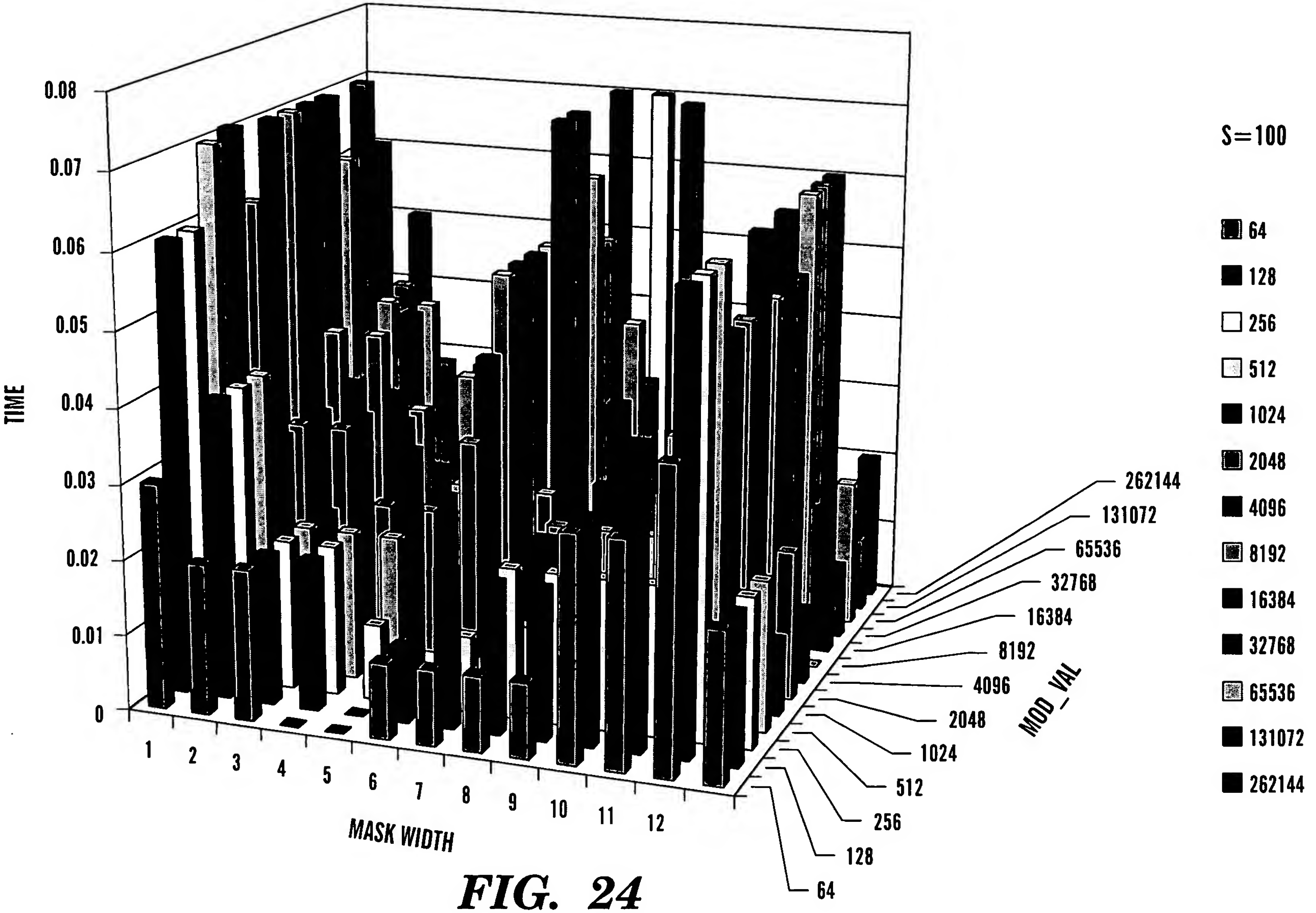

S=100
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
TIME
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
MASK WIDTH
1
2
3
4
5
6
7
8
9
10
11
12
MOD_VAL
262144
131072
65536
32768
16384
8192
4096
2048
1024
512
256
128
64

FIG. 24